UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Uroš Vonta

Aplikacijsko ogrodje v Javi za hiter razvoj aplikacij s porazdeljenimi podatkovnimi viri

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2016

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Uroš Vonta

Aplikacijsko ogrodje v Javi za hiter razvoj aplikacij s porazdeljenimi podatkovnimi viri

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: viš. pred. dr. Aljaž Zrnec

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Programske rešitve se morajo velikokrat povezati na različne podatkovne vire, iz njih prebrati določene podatke, jih agregirati in posredovati naprej. To velja predvsem za sistemske integracije, kjer se podatki pogosto nahajajo na povsem različnih lokacijah in platformah. Pri tem se pojavi problem, da morajo razvijalci velik del časa nameniti kodi za povezovanje na posamezne podatkovne vire. Zasnujte programsko ogrodje, ki bo razvijalcem omogočalo, da se osredotočijo na pisanje poslovnih pravil za svoje aplikacije, dostop do podatkov pa bi potekal po enotnem APIju, ne glede na izbrane vire. Prednost takega ogrodja za razvoj aplikacij bi bila tudi popolna prenosljivost podatkov na druge vire, saj se sama poslovna logika zaradi menjave podatkovnega vira nič ne spremeni.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Uroš Vonta sem avtor diplomskega dela z naslovom:

Aplikacijsko ogrodje v Javi za hiter razvoj aplikacij s porazdeljenimi podatkovnimi viri (angl. Java application framework for rapid development of applications with distributed data sources)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom viš. pred. dr. Aljaža Zrneca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 28. aprila 2016 Podpis avtorja:

Zahvaljujem se mentorju viš. pred. dr. Aljažu Zrnecu za usmerjanje in napotke pri izdelavi diplomske naloge. Predvsem pa sem hvaležen svoji dragi Aniti za ljubezen, moralno in čustveno podporo ob izdelavi diplomske naloge.

Kazalo

Slike

Seznam uporabljenih kratic

kratica	angleško	slovensko
ACID	Atomicity, Consistency, Isolation, Durability	atomarnost, konsistentnost, izolacija, trajnost
API	Application Programming Interface	programski vmesnik
CRUD	Create, Read, Update and Delete	kreiranje, branje, posodabljanje in brisanje
CSS	Cascading Style Sheets	kaskadne stilske podloge
DOM	Document Object Model	dokumentno objektni model
DSL	Domain-Specific Language	domensko specifičen jezik
GUI	Graphical User Interface	grafični uporabniški vmesnik
HTML	HyperText Markup Language	jezik za označevanje nadbesedila
HTTP	Hypertext Transfer Protocol	hipertekstovni prenosni protokol
ID	Identifier	identifikator
JAR	Java Archive	javanski arhiv
JDBC	Java Database Connectivity	javanska fasada za dostop do podatkovnih baz
JDK	Java Development Kit	javanski razvojni komplet
JSON	JavaScript Object Notation	JavaScript objektna notacija
JVM	Java Virtual Machine	javanski navidezni stroj
NoSQL	No Structured Query Language	nestrukturiran poizvedovalni jezik
REST	Representational State Transfer	predstavitvena arhitektura za prenos podatkov

SOAP	Simple Object Access Protocol	protokol za izmenjavo podatkov med spletnimi storitvami
SQL	Structured Query Language	strukturirani povpraševalni jezik za delo s podatkovnimi bazami
VCS	Version Control Systems	sistem za nadzor različic
WSDL	Web Services Description Language	jezik za opis spletnih storitev
XML	Extensible Markup Language	razširljiv označevalni jezik

Povzetek

Naslov: Aplikacijsko ogrodje v Javi za enostaven razvoj aplikacij s porazdeljenimi podatkovnimi viri

S hitrim razvojem informacijske tehnologije in ogromnim naborom aplikacij se pojavlja potreba po programskih rešitvah, ki bi na enem mestu združevale podatke iz različnih virov. Cilj diplomske naloge je bil narediti aplikacijsko ogrodje v programskem jeziku Java, ki bi omogočal hitro izdelavo programskih rešitev, ki uporabljajo enega ali več različnih podatkovnih virov. Razvijalci programske opreme bi z uporabo takega ogrodja potrebovali manj časa za razvoj aplikacije, saj bi se lahko bolj posvetili poslovnim pravilom aplikacije, kot pa programski kodi, ki skrbi za dostop do podatkov. Ogrodje je izdelano tako, da ponuja dve stvari; preprost način za definiranje poslovne logike vsake entitete, ki nastopa v aplikaciji, ter enoten API za dostop do podatkov, ne glede na tip podatkovnega vira. S takim načinom razvoja postane programska koda bolj razumljiva, berljiva in lažja za vzdrževanje.

Ključne besede: aplikacijsko ogrodje, podatkovni viri, Java, razvoj programske opreme, testiranje, združevanje podatkov

Abstract

Title: Java application framework for easy development of applications with distributed data sources

With rapid development of information technology and a huge range of applications there is a need for software solutions that offers centralized data from different sources. The aim of the thesis was to make application framework in the Java programming language, which would enable the rapid creation of software solutions using one or more different data sources. With this kind of framework, software developers would need less time to develop their applications. They would be more focused on business rules, rather they would spent time on developing software code for data persistency. The framework is designed in such a way that offers two things; a simple way to define business logic of each entity and a unified API for accessing data regardless the type of data source. This kind of software development makes source code more understandable, readable and maintainable.

Keywords: application framework, data sources, Java, software development, testing, data aggregation

Poglavje 1 Uvod

V današnjem času je hiter dostop do informacij različnega značaja ključen za uspešno delovanje tako posameznika kot tudi podjetja. Predvsem je pomemben hiter dostop do ključnih informacij, ki pa so praviloma razkropljene po različnih lokacijah in v različnih sistemih. Vse več podjetij se zaveda, da je za dobro poslovanje in konkuriranje na trgu potrebna dobra in predvsem učinkovita informacijska podpora. Veliko podjetij ima namreč težave z razkropljenostjo podatkov po različnih sistemih, kar vodi do neučinkovitega dela zaposlenih, saj morajo za svoje delovanje preskakovati iz sistema v sistem in iskati podatke, ki so za njih v danem trenutku pomembni. Težave pri taki situaciji so tudi:

- pozabljanje uporabniških imen in gesel za različne aplikacije,
- potek veljavnosti gesel (če aplikacija zahteva redno menjavanje gesel),
- potrebna fizična zamenjava klienta (ker so aplikacije napisane za določen operacijski sistem, ne delujejo v vseh brskalnikih),
- redundanca podatkov (različni sistemi v podjetju potrebujejo enake podatke – podatki o zaposlenih, o partnerjih ipd.),
- nasprotujoči si podatki (sprememba e-poštnega naslova zaposlenega je vnesena v en sistem v drugem pa je še stara vrednost).

Zaradi prej omenjenih težav vse več podjetij naroča razvoj aplikacij, ki združuje različne sisteme v nekakšne portale, kjer imajo zaposleni dostop do vseh ključnih podatkov na enem mestu, kar zelo pripomore k splošni produktivnosti.

V diplomski nalogi se bomo osredotočili na poenostavitev razvoja aplikacij, ki uporabljajo podatke iz različnih podatkovnih virov. V nadaljevanju bomo opredelili zahteve ogrodja ter njegov namen. Omenili bomo tudi vse uporabljene tehnologije s katerimi je bilo razvito ogrodje, ter katere knjižnice so potrebne za njegovo delovanje. Nekaj besed bo namenjenih tudi orodjem, ki so bila uporabljena pri razvoju. Našo rešitev bomo opisali s stališča funkcionalnosti in arhitekture. Na koncu bomo zapisali še naše sklepne ugotovitve ter kako smo zadovoljni s

končnim produktom. Navedli bomo tudi njegove prednosti in slabosti ter kaj bi bilo v prihodnosti še smiselno dograditi, oziroma spremeniti.

Poglavje 2 Opredelitev zahtev in funkcionalnosti

2.1 Namen ogrodja

Namen diplomske naloge je razviti aplikacijsko ogrodje v programskem jeziku Java, ki bi omogočal razvijalcem programske opreme hiter razvoj predvsem takih aplikacij, ki združujejo podatke iz različnih podatkovnih virov. To seveda ni pogoj, saj je lahko podatkovni vir tudi en sam. Namen ogrodja je tudi ta, da se razvijalec ne ukvarja s kodo, ki se dejansko povezuje na podatkovne vire, ampak se osredotoči na razvoj poslovnih pravil. Kako bo aplikacija brala oziroma zapisovala v podatkovni vir, pa je stvar tega ogrodja.

Ogrodje bo pripomoglo k hitrejšemu razvoju predvsem takih aplikacij, ki predstavljajo sistemske integracije v podjetjih, kjer si želijo aplikacijo, ki bi združevala podatke iz navadno že obstoječih sistemov. Osnovna enota ogrodja je entiteta, ki predstavlja neko logično enoto (na primer v relacijskih podatkovnih bazah je vsaka tabela svoja entiteta). Entiteta je lahko oseba, račun, partner, organizacijska enota, delovno mesto, itd. Ogrodje bo glede na entiteto avtomatsko izbralo ustrezno vrsto povezave na podatkovni vir, od koder bo podatke bralo oziroma jih vanj zapisovalo.

En izmed namenov ogrodja je tudi ta, da se mehanizem za manipulacijo in dostop do podatkov skrije – uporabimo princip »črne škatle«. To pomeni, da razvijalcem ni potrebno vedeti, kako aplikacija v ozadju dejansko dostopa do podatkov. Poznati morajo le preprost API, ki jim omogoča izvajati CRUD operacije, ne glede na tip podatkovnega vira. Tak način zelo poenostavi razvoj aplikacij, ki uporabljajo zunanje podatkovne vire, in poveča število razvijalcev z zadostnim znanjem, saj ne potrebujejo znanja za delo s podatkovnimi bazami, ampak je osnovno znanje programskega jezika Java popolnoma dovolj.

2.2 Zahteve

V tem poglavju se bomo osredotočili na zahteve, katerim naj bi naše ogrodje zadostilo. Ali je bilo zadoščeno vsem zahtevam, pa bo opisano v nadaljevanju diplomske naloge, kjer bodo opisane vse implementirane funkcionalnosti. Glavne zahteve našega ogrodja so:

1. Ogrodje naj bo napisano v programskem jeziku Java, pomožni dodatki pa naj bodo napisani v jeziku Groovy, ki se jih lahko zažene kot skripte med izvajanjem.

2. Glede na današnjo popularnost REST APIja bomo kot osnovo za vse razrede entitet vzeli asociativna polja, ki jih v Javi predstavljajo objekti tipa Map. Ti objekti so naravno najbolj podobni notaciji JSON, kjer gre predvsem za objekte tipa ključ/vrednost ali angleško key/value objects. Tako bomo imeli v okviru ogrodja opravka po večini le z objekti, ki razširjajo objekte Map, ter s seznami, ki vsebujejo prej omenjene objekte. S tem tudi zelo poenostavimo programiranje v programskem jeziku Groovy, saj lahko uporabimo njegov izvirni jezik za delo z objekti tipa Map in List, ki je zelo preprost za uporabo.
3. Glede na to, da bo ogrodje omogočalo manipulacijo podatkov po različnih podatkovnih virih, mora vsak objekt vsebovati atribut z imenom entitete, na podlagi katerega bo ogrodje »vedelo«, kam naj objekt zapiše oziroma od kod naj ga bere. Ta podatek bo v vsakem objektu (vsak objekt je asociativno polje, kot smo to zapisali v 2. točki) zapisan pod ključem »entity«. Če atribut »entity« ne bo shranjen v samem zapisu, bo ogrodje poskrbelo, da se ga bo ob branju iz podatkovnega vira dodalo v prebran objekt (v relacijskih podatkovnih bazah ga v samih tabelah verjetno ne bo, ampak bo entity kar ime tabele, v NoSQL podatkovnih bazah, pa bo verjetno del samega objekta v bazi) ter odstranilo pred pisanjem v podatkovni vir.
4. Vsaka entiteta mora imeti tudi svoj opis, ki se bo nahajal v datotekah tipa *descriptor*. V vsakem takem deskriptorju bo opis vseh atributov določene entitete ter še kakšne druge lastnosti entitet, ki so potrebne za delovanje ogrodja. Tukaj bodo definirani tudi objekti, ki bodo definirali katere pogoje lahko uporabimo pri iskanju. Vzemimo za primer izmišljeno entiteto Dokument v neki relacijski podatkovni bazi, ki se nahaja v tabeli Dokument z naslednjimi atributi; ID, naziv, vrsta, status in vsebina. Če ima tabela Dokument izdelan indeks po stolpcu status, lahko v deskriptorju entitete definiramo filter, ki vsebuje le atribut status. Če znotraj ogrodja prožimo iskanje po atributu status, bo v primeru izmišljene entitete Dokument sprožena poizvedba z WHERE pogojem, ker smo definirali filter za atribut status. Če filtra ne bi definirali, bi se sprožila poizvedba brez WHERE pogoja, ki bi vrnila vse zapise in bi ogrodje kasneje filtriralo po atributu status.
5. Glavna datoteka z nastavitvami aplikacije bo vsebovala seznam vseh uporabljenih entitet v aplikaciji ter nastavitve za dostop do vseh podatkovnih virov s seznamom entitet, ki spadajo vanj. Tu se bodo nahajale tudi ostale globalne nastavitve same aplikacije, ki bo razvita z uporabo našega aplikacijskega ogrodja.
6. Ogrodje mora zagotoviti enoten API za CRUD operacije (kreiranje, branje, posodabljanje in brisanje) ne glede na vrsto podatkovnega vira. S tem bo zagotovljena transparentnost pri dostopu do različnih podatkovnih virov, kar bo omogočalo prenosljivost podatkov na drug podatkovni vir, pri čemer se sama koda aplikacije nič ne

spremeni. Spremeni se le vsebina nastavitvene datoteke, kjer bodo definirani drugi podatkovni viri. Zaradi te lastnosti bo naše ogrodje lahko tudi dobra osnova za testiranje prepustnosti različnih podatkovnih baz, saj bomo lahko enako kodo uporabili na različnih bazah in nato primerjali rezultate.

Poglavje 3 Pregled tehnologij in orodij

3.1 Uporabljene tehnologije

V nadaljevanju se bomo posvetili tehnologijam, ki smo jih uporabili za razvoj ogrodja. Osnova za ogrodje je programski jezik Java, za dodatne nastavitvene in druge opisne datoteke pa smo uporabili programski jezik Groovy, saj je zelo primeren za pisanje DSL skriptnih datotek, ki se jih lahko interpretira med samim izvajanjem aplikacije brez predhodnega prevajanja v binarno kodo.

3.1.1 Java

Java je danes eden izmed najbolj popularnih programskih jezikov za razvoj programske opreme. Je sodoben, visoko-nivojski, objektno usmerjen programski jezik. Programska oprema napisana v Javi nas dandanes obkroža praktično na vsakem koraku. Nahaja se na mobilnih telefonih (npr. Android, JavaME), v vgrajenih sistemih (ang. Embedded System), na namiznih računalnikih kot samostojne aplikacije, na spletnih strežnikih itd.

Prva verzija Jave je bila objavljena leta 1995 pod okriljem podjetja Sun Microsistems. Na začetku je Java slovela kot malce počasnejša in bolj pomnilniško potratna glede na ostale programske jezike (kot npr. C, C++). Temu botruje predvsem vmesni navidezni stroj, ki seveda zahteva nekaj dodatnih resursov. Programska koda napisana v Javi se namreč ne prevede v binarno izvršljivo kodo, ki se požene direktno na operacijskem sistemu, ampak se prevede v vmesno kodo, ki se izvaja na javanskem navideznem stroju oziroma na JVMju. Danes je ta razlika z ostalimi programskimi jeziki, ki ne uporabljajo navideznega stroja, praktično zanemarljiva, tako da je programska oprema napisana v Javi popolnoma konkurenčna z ostalimi aplikacijami, napisanimi v ostalih programskih jezikih [10, 12].

Velika prednost Jave je prenosljivost med različnimi operacijskimi sistemi, saj se izvorna koda napisana v Javi ne prevede direktno v binarno kodo, ki se lahko izvaja le na določenem operacijskem sistemu, ampak se prevede v vmesno kodo, ki se izvaja na javanskem navideznem stroju. Tako je potrebno za vsak operacijski sistem napisati specifičen JVM, aplikacije v Javi pa so napisane le enkrat in lahko tečejo na kateremkoli JVM navideznem stroju. Trenutni lastnik licence za uradno verzijo Jave je Oracle Corporation.

3.1.2 Groovy

Programski jezik Groovy je prav tako kot Java objektno usmerjen programski jezik, saj bazira na Javi. Programska koda napisana v Groovyju se tako kot javanske datoteke prevede v javanske binarne datoteke (to so datoteke *.class), ki se izvajajo na JVMju. To pomeni, da lahko razvijamo aplikacijo, ki vsebuje tako javanske kot groovyjeve razrede. Od verzije 2.0 naprej, ki je bila predstavljena javnosti 2012, se Groovy lahko uporablja tudi kot skriptni jezik, ki se prevede in izvede med samim izvajanjem aplikacije. Groovyjeva sintaksa je podobna javanski, tako da lahko vsak programer, ki pozna Javo, hitro osvoji tudi programski jezik Groovy. Skoraj vsak javanski razred je tudi veljaven Groovy razred. Groovyjeva sintaksa je tudi veliko bolj kompaktna kot javanska, saj ne potrebuje vseh elementov, ki jih potrebuje Java. Zaradi teh lastnosti je Groovy primeren za hiter razvoj programske opreme, saj je koda preprostejša od Jave, vendar pa je lahko zaradi tega malenkost manj učinkovita. Groovy ima obilico lastnosti in gradnikov za lažje in hitrejše programiranje, ki jih programerji pogrešamo pri programskem jeziku Java. Naj omenimo le nekaj najbolj zanimivih [2, 5]:

- GroovyBean – Groovy za vse atribute razreda avtomatsko generira GET in SET metode;
- operator »@« - dostop do privatnih atributov razreda;
- closure – to so kodni bloki, ki jih lahko definiramo in nato izvajamo na poljubnem mestu v kodi. Lahko jih tudi posredujemo kot atribut funkcije;
- vgrajena podpora za regularne izraze;
- vgrajena podpora za branje in kreiranje JSON in XML objektov (sprehajanje po objektih v DOM notaciji);
- polimorphic iteration – enake funkcije za iteriranje čez različne objekte (seznami, nizi, polja);
- operator »?« za preprečitev NullPointerException napake (npr. mojObjekt?.izvedi() – metoda izvedi() v spremenljivki mojObjekt se ne bo klicala, če je mojObjekt enak null);
- meta programiranje – obstoječim objektom lahko programsko dodajamo nove funkcionalnosti.

3.2 Uporabljene knjižnice

V tem razdelku bomo nekaj besed namenili nekaterim najpomembnejšim knjižnicam, ki smo jih uporabili pri razvoju naše rešitve. Za hiter in učinkovit razvoj programske opreme se razvijalci poslužujemo že napisane programske kode za standardna opravila. Slednjo običajno najdemo kot samostojen kos programske kode na spletu ali pa uporabimo kakšno izmed prosto dostopnih knjižnic. Tako ne izgubljamo dragocenega časa in se lahko bolj posvetimo dejanskim

problemom in zahtevam, namesto da se ukvarjamo s tem, kako na primer prebrati vsebino datoteke z datotečnega sistema, kar je bilo že ničkolikokrat sprogramirano.

3.2.1 Apache Maven

Apache Maven je orodje, ki poskrbi za avtomatizacijo izdelave projekta. Uporablja se za prevod javanske kode, za avtomatski zagon testov programskih enot, skrbi da ne prihaja do cikličnih odvisnosti med moduli aplikacije, itd. Z njim lahko izdelamo verzijo aplikacije in tudi namestimo aplikacijo v razvojno okolje. Maven poskrbi tudi za avtomatski prenos potrebnih knjižnic iz za to namenjenih strežnikov, tako da se razvijalcem ni potrebno ukvarjati z dodajanjem knjižnic na projekt, da bi se koda lahko ustrezno prevedla. Zgrajen je na arhitekturi, ki temelji na uporabi vtičnikov, kar ga naredi izredno razširljivega. Teoretično to pomeni, da lahko vsakdo napiše svoj vtičnik, ki ga potem uporabi v življenjskem ciklu razvoja programske opreme [4, 8].

3.2.2 JUnit

Knjižnica JUnit implementira ogrodje za testiranje enot (ang. Unit testing). Je ena izmed največkrat, če ne največkrat, uporabljena knjižnica v javanskih aplikacijah. Testiranje enot je dobra praksa pri razvoju programske opreme, saj lahko tako sproti med razvojem ulovimo kar se da veliko napak oziroma tako imenovanih bugov. Unit testi so testi programskih enot, ki med seboj niso odvisni. Dobra praksa razvoja programske opreme je tudi avtomatsko zaganjanje vseh testov programskih enot ob prenosu kode v sistem za nadzor različic (ang. Version control system). Za to so po navadi zadolženi tako imenovani Continuous integration sistemi. Taki sistemi večkrat na dan (odvisno od nastavitev) k sebi prenesejo zadnjo verzijo izvorne kode, jo prevedejo in zaženejo vse teste enot. Ob neuspešnem prevodu kode ali ob napakah pri poganjanju testov lahko tako nemudoma obvestijo razvijalca, ki je zadnji prenesel kodo v sistem za nadzor različic. Tako je napaka kar najhitreje odpravljena, saj razvijalec po navadi ve, kaj je nazadnje spreminjal in kje bi lahko prišlo do napake. Tako hitreje odpravi napako, kot če bi moral napako odpravljati kasneje, ko je bilo spremenjene veliko več programske kode [6].

3.2.3 Jackson

Jackson je ena izmed najbolj znanih in najbolj učinkovitih javanskih knjižnic za pretvorbo objektov JSON. Glede na to, da je REST API dandanes zelo popularna oblika za komunikacijo med sistemi ali pa med strežniškim delom aplikacije in GUIjem, smo za osnovo našega ogrodja vzeli asociativna polja (angl. Map), saj se narava teh objektov najbolj približa JSON notaciji. Spletni servisi, ki za komunikacijo uporabljajo REST API so tudi učinkovitejši, bolj kompaktni in lažje berljivi, kot recimo komunikacija po protokolu SOAP, ki uporablja bolj okorno ter

prostorsko bolj potratno strukturo XML. Prednost protokola SOAP je le v večji standardizaciji spletnih servisov, saj so vhodni in izhodni objekti točno definirani s strukturo WSDL, pri REST protokolu pa tega opisa spletnega servisa nimamo, tako da se morajo razvijalci pri razvoju opirati bolj na poizkušanje.

3.2.4 Slf4j

Dobra praksa izdelave programske opreme je tudi učinkovito izpisovanje sporočil med delovanjem aplikacije. Za to smo uporabili knjižnico Slf4j, ki je nekakšna fasada oziroma vmesnik med našo programsko kodo in knjižnico, ki dejansko izpisuje sporočila na nek definiran izhod. Na ta način smo naredili ogrodje bolj neodvisno in prenosljivo, saj je zmožno uporabiti knjižnico za izpis na gostujočem sistemu. Slf4j zna uporabljati naslednje knjižnice, ki so namenjene izpisovanju obvestil in napak; java.util.logging, logback, log4j, tinylog,… To pomeni, da če našo aplikacijo namestimo na nek aplikacijski strežnik, ki za izpise uporablja log4j, potem naše kode ni potrebno nič več spreminjati in prilagajati za delo z log4j knjižnico, ampak bo ogrodje Slf4j sporočila iz naše aplikacije avtomatsko preusmerila v knjižnico log4j, ki bo poskrbela za dejanski izpis sporočila na definiran izhod.

3.2.5 Apache Commons knjižnice

Knjižnice Apache Commons implementirajo najbolj pogoste operacije, ki jih potrebujemo pri mnogih aplikacijah. Knjižnico Apache Commons-IO smo tako uporabili za vhodno-izhodne akcije, saj vsebuje funkcije za delo s podatkovnimi tokovi, datotekami, primerjalniki, itd. Za delo s klici HTTP smo uporabili knjižnico Apache HttpClient, ki implementira funkcije za delo s protokolom HTTP. Za razne standardne akcije pa smo uporabili Apache Commons-lang3 knjižnico, ki implementira dodatne funkcije javanskega paketa java.lang, ki jih Java sama po sebi ne zagotovi. To so razne funkcije za manipulacijo z znakovnimi nizi oziroma Stringi, za delo z numeričnimi operatorji, sočasnostjo, sistemskimi spremenljivkami, itd. [11].

3.3 Razvojna orodja

V tem razdelku bomo opisali katera razvojna orodja smo uporabili pri razvoju našega ogrodja. Brez naprednih razvojnih orodij si dandanes ne predstavljamo razvoja aplikacij, saj bi bilo obvladovanje množice besedilnih datotek s programsko kodo praktično nemogoče oziroma bi usklajevanje le-teh vzelo ogromno časa. Kompleksnost pri upravljanju datotek s programsko kodo se še dodatno poveča, če aplikacijo razvija več razvijalcev hkrati, saj je takrat potrebno usklajevati različne verzije ene datoteke, saj lahko vsak razvijalec naredi popravek v isti datoteki.

3.3.1 Eclipse IDE

Eclipse je eno izmed najbolj popularnih integriranih odprtokodnih razvojnih okolij. Je zelo prilagodljivo okolje, saj se uporablja za razvoj aplikacij v velikem številu programskih jezikov in tehnologij kot so Java, C/C++, PHP, Android, JavaScript, HTML, CSS, XML,… Eclipse deluje na principu vtičnikov, ki si jih lahko vsak posameznik naloži glede na potrebe. Vsak lahko razvije tudi vtičnik po svoji meri. Če želi, ga lahko potem deli z ostalimi razvijalci tako, da ga objavi na tako imenovanem Marketplaceu. Za razvoj v programskem jeziku Java je Eclipse najbolj popularno brezplačno razvojno okolje, zato smo ga uporabili za razvoj našega ogrodja [9].

3.3.2 Sistemi za nadzor različic

Vsaka malo večja aplikacija, ki ni stvar nekaj sto vrstic kode, potrebuje za učinkovito upravljanje programske kode sistem za nadzor različic ali z angleško kratico VCS. Osnovni namen takih sistemov je, da ima v nekem centralnem repozitoriju zadnjo veljavno oziroma delujočo programsko kodo. Vsak razvijalec si nato k sebi prenese kopijo, ki jo popravlja, dokler ni zadovoljen s spremembami. Ko je s spremembami zadovoljen, jih sinhronizira s centralnim repozitorijem, kamor se potem prenesejo spremembe in si jih lahko ostali člani razvojne ekipe zopet prenesejo k sebi.

Če takega orodja ne uporabimo, kaj hitro naletimo na težave, ko popravimo že kar nekaj datotek s programsko kodo, potem pa ugotovimo, da teh sprememb ne bi radi obdržali in bi radi vse spremembe zavrgli. Tukaj pride sistem za nadzor različic prvič do izraza, saj lahko z enim ukazom vse spremembe, ki smo jih naredili od zadnjega prenosa v centralni repozitorij, enostavno zavržemo.

Druga zelo velika prednost takih sistemov se pokaže, ko isto aplikacijo razvija več kot en razvijalec. Če takega sistema ne bi uporabili, bi si morali razvijalci nenehno pošiljati spremenjene datoteke med seboj, pri čemer je neizogibno, da bo nekdo slej ali prej povozil spremembe nekoga drugega. Sistem za nadzor različic pa take primere prepreči, saj ugotovi ali sta dva razvijalca popravljala isto datoteko. Tako mora drugi razvijalec najprej sinhronizirati svojo datoteko z datoteko, ki jo je pred njim v centralni repozitorij prenesel prvi razvijalec in šele na to jo lahko prenese tudi on, s čimer je ohranil spremembe prvega razvijalca.

Tretja velika prednost takih sistemov pa so posnetki celotne programske kode za določeno časovno rezino oziroma tako imenovani tagi. To pride dejansko prav takrat, ko izdamo verzijo aplikacije in želimo imeti v prihodnosti možnost vpogleda v kodo, ki je bila aktualna v času

izdaje verzije. Tako imamo možnost, da popravimo kakšno kritično napako določene verzije in izdamo verzijo s popravki ne glede na to, da imamo razvitih že nekaj verzij naprej.

Še ena velika prednost takih sistemov pa je ta, da lahko naredimo novo vejo ali branch trenutno aktualne programske kode. To pride zelo prav, ko želimo razviti neko dodatno logiko oziroma funkcionalnost in še ne vemo ali se bo obnesla ali ne, ali pa preprosto želimo narediti verzijo kode z nekaterimi spremembami glede na glavno vejo. Dodatno vejo lahko kasneje združimo z glavno vejo, jo preprosto pobrišemo, če rešitev ne ustreza, ali pa jo enostavno vodimo kot dodatno vejo s posebnostmi [3].

Poglavje 4 Razvoj in opis ogrodja

4.1 Osnovni opis

Aplikacijsko ogrodje, ki smo ga razvili je napisano v programskem jeziku Java, kot smo to omenili že v zahtevah za razvoj ogrodja. Uporabili smo verzijo Java Standard Edition 8. Nastavitvena datoteka ter ostale datoteke z opisi entitet, ki nastopajo v ogrodju, so DSL skripti, napisani v programskem jeziku Groovy. Uporabili smo trenutno najnovejšo različico Groovyja, to je verzija 2.4.3. Celotno ogrodje se lahko zapakira v eno knjižnico JAR, ki se jo potem ob razvijanju nove aplikacije le doda v projekt. To se lahko stori ročno ali s pomočjo orodja Maven, kjer v konfiguracijski datoteki pom.xml definiramo ustrezno odvisnost (angl. dependency), katera postavi knjižnico z našim ogrodjem na programsko pot, da je dostopna ob izvajanju in prevajanju kode. Poleg knjižnice z ogrodjem morajo biti na projektu za uspešno delovanje in prevod programske kode dodane tudi vse ostale potrebne knjižnice, ki so bile uporabljene v samem ogrodju in so opisane v prejšnjih poglavjih. Ogrodje se lahko uporabi tako za razvoj namiznih aplikacij, za spletne aplikacije, ki tečejo na aplikacijskih strežnikih, za razvoj mikrostoritev in še mnogo drugih vrst aplikacij.

Kot smo definirali že v zahtevah, ogrodje omogoča poganjanje iste aplikacije nad katerimkoli podprtim podatkovnim virom. Ta funkcionalnost naredi naše ogrodje zelo uporabno iz dveh vidikov:

1. Izdelamo lahko aplikacije, ki niso odvisne od podatkovnega vira in jih lahko kadarkoli hitro in brez dodatnih posegov v programsko kodo preselimo na drug podatkovni vir. Spremeniti je potrebno le nastavitveno datoteko, kjer definiramo vrsto podatkovnega vira in podatke za povezavo nanj.
2. Ogrodje lahko uporabimo za primerjavo učinkovitosti različnih podatkovnih baz, saj testni scenarij napišemo enkrat, potem pa le s pomočjo konfiguracij test poženemo nad različnimi bazami podatkov. V sklopu diplomske naloge je bila narejena zelo preprosta aplikacija, ki ilustrira delovanje našega aplikacijskega orodja v praksi, in sicer tako, da vsebuje JUnit teste za štiri različne podatkovne baze.

V nadaljevanju bomo opisali, kako uporabiti razvito ogrodje za razvoj aplikacij, kakšna je arhitektura ogrodja, katere funkcionalnosti so razvite, itd. Poleg ogrodja smo zraven razvili tudi preprosto aplikacijo, ki demonstrira uporabo ogrodja.

4.2 Funkcionalnosti

4.2.1 Enoten API za dostop do podatkov

Kot smo definirali že v zahtevah, je glavna funkcionalnost našega ogrodja enoten oziroma transparenten API za dostop do podatkov. To pomeni, da se razvijalcu aplikacije ni potrebno ukvarjati s tem, od kod in na kakšen način bo podatke bral iz podatkovnega vira, ampak da za to poskrbi ogrodje samo. Vse CRUD operacije se izvedejo preko razreda Model. Razred Model poskrbi, da gre vsaka entiteta v ustrezen podatkovni vir, ter tudi da se izvedejo ustrezna pravila za posamezno entiteto. S tem je poskrbljeno, da že samo ogrodje poskrbi za izvajanje pravil, ki so potrebna za pravilno delovanje celotnega sistema. Razvijalci morajo za vsako entiteto le kreirati pripadajoči razred s pravili, Model pa jih med samim izvajanjem uporabi na ustreznih mestih.

Razred Model izpostavlja naslednje javne funkcije:

- read – branje zapisa po IDju;
- readAll – branje seznama zapisov glede na podan seznam IDjev;
- find – iskanje zapisa katerega atribut ustreza podani vrednosti (če iskalna poizvedba vrne več kot en rezultat, Model proži napako);
- findAll – iskanje vseh zapisov, ki ustrezajo iskalnem pogoju;
- create – kreiranje zapisa v podatkovnem viru;
- update – posodobitev zapisa v podatkovnem viru;
- delete – brisanje zapisa iz podatkovnega vira.

Vsaka izmed funkcij ima »pred« in »po« pravila. Če se »pred« pravilo ne izvede uspešno, sploh ne pride do interakcije s podatkovnim virom. Vsaka entiteta ima lahko definirana svoja pravila, ki so definirana v razredu z nazivom <ime_entitete>_Rules.java v paketu, ki ga definiramo v nastavitveni datoteki. Če razvijamo aplikacijo z več uporabniki, ki imajo lahko različne pravice, lahko v teh razredih s pravili za vsako entiteto posebej definiramo, katere pravice so potrebne za kreiranje, branje, urejanje ali brisanje. Pri uporabi funkcije findAll (iskanje vseh zapisov neke entitete ali tistih, ki ustrezajo podanemu pogoju) lahko na primer v »po« pravilih glede na pravice uporabnika iz rezultata odstranimo le del ali pa vse zapise, če ugotovimo, da uporabnik nima zadostnih pravic za ogled najdenih entitet. Po branju entitete lahko dodamo kakšna

dodatno izpeljana polja, lahko pokličemo morebitni zunanji servis, pošljemo e-poštno sporočilo, itd.

4.2.2 Entitete pripravljene za REST API

Vsaka entiteta v sistemu je podrazred razreda ModelObject (ModelObject implementira vmesnik Map), ki predstavlja asociativno polje. Taka oblika je zelo primerna za transformacijo v objekt tipa JSON, ki je dandanes zelo priljubljena oblika za komunikacijo med različnimi sistemi, aplikacijami, moduli, med strežniškim delom aplikacije in uporabniškim vmesnikom in še bi lahko naštevali.

Vsak razred, ki predstavlja entiteto, mora razširjati razred *Entity* iz ogrodja. Entity razred pa razširja osnovni objekt ogrodja ModelObject, ki ima prekrito funkcijo toString() tako, da trenutni objekt pretvori direktno v niz tipa JSON. Ta funkcionalnost naše ogrodje naredi še bolj prijazno, če razvijamo aplikacijo, ki uporablja REST API, sej je to že del samega ogrodja. V takem primeru morajo razvijalci programske opreme nad entiteto le še poklicati funkcijo toString(), ki objekte spremeni v nize tipa JSON.

Model ogrodja vedno vrača sezname zapisov v objektu ModelObjectList, ki prav tako prekriva funkcijo toString() tako, da se seznam pretvori v niz, ki predstavlja seznam v obliki JSONa. To omogoča pretvorbo v veljaven niz tipa JSON ne glede na to, ali imamo opravka s seznami ali z objekti (seznami so lahko tudi gnezdeni v samem objektu).

Ogrodje omogoča tudi obraten proces pretvorbe objektov v nize tipa JSON. Za namen pretvorbe objektov v JSONe in obratno je izdelan statični razred JsonTools.

4.2.3 Uporaba načrtovalskega vzorca Command pattern

Ogrodje je bilo že v začetku načrtovano tako, da se lahko kasneje hitro nadgradi z novimi funkcionalnostmi. Vsak zahtevek spremlja objekt Command, ki nosi metapodatke o zahtevku. Objekt Command se kreira na začetku zahtevka in ga spremlja vse do konca. Predvsem je bilo mišljeno, da Command nosi s seboj tudi podatke o seji, če je aplikacija napisana za več uporabnikov, kakšne dodatne vhodne parametre, itd. Na ta način bi lahko potem razredi s poslovnimi pravili avtomatsko preverili ali ima trenutno prijavljeni uporabnik pravico videti prebrani objekt oziroma seznam objektov ali ima pravico, da ga posodobi, pobriše ali kreira.

Objekt Command trenutno vsebuje naslednji nabor podatkov:

- naziv entitete za katero je bila sprožena zahteva;

- entiteto ali seznam entitet, ki so bile prebrane, oziroma so namenjene zapisu v podatkovni vir;
- trenutni jezik uporabnika oziroma aplikacije, saj se na podlagi tega izpiše obvestilo o morebitni napaki iz ustrezne datoteke s prevodi;
- podatke, ki so potrebni ob branju zapisa ali seznama zapisov iz podatkovnega vira (ID zapisa ali naziv filtra ter vrednost, po kateri filtriramo).

Ta princip načrtovalskega vzorca z objektom Command je zelo primeren za pisanje dnevnika zahtevkov, saj imamo v Commandu vse podatke, s katerimi lahko ugotovimo kako in s kakšnimi parametri je bil zahtevek kreiran. Kjerkoli v programski kodi pride do izjeme, lahko zraven opisa napake zapišemo tudi vsebino objekta Command, saj le-ta vedno spremlja zahtevek od začetka do konca. Izpis objekta Command nam poleg opisa napake lahko kasneje zelo olajša delo ob ugotavljanju vzroka za napako, saj vidimo za kakšen tip zahtevka je šlo in kakšne so bile vrednosti vstopnih parametrov. Tako lahko na preprost način rekonstruiramo scenarij ter ponovimo zahtevek z enakimi parametri, s čimer privedemo sistem do ponovitve napake [1].

4.2.4 Dvonivojski izpis napak s podporo za večjezičnost

V ogrodje smo vgradili tudi sistem za dvonivojsko izpisovanje sistemskih napak, ki se zgodijo med delovanjem aplikacije. To pomeni, da se sporočila napak delijo na dva dela; izpis originalne napake, ki se ponavadi zapiše v dnevnik napak ter sporočilo, namenjeno uporabniku aplikacije. Čeprav te funkcionalnosti ni bilo v zahtevah za razvoj ogrodja, smo jo vseeno razvili, saj se nam je zdela zelo uporabna. Tako smo naše ogrodje pripravili tudi za razvoj večjezičnih aplikacij, kar zelo poveča samo vrednost ogrodja.

Za ta namen je bil izdelan osnovni razred AppException, ki predstavlja napako v našem ogrodju. AppException razred razširja razred java.util.Exception. Glavni dve razliki med osnovno javansko napako in napako AppException sta dodatna funkcija za izpis obvestila za uporabnika. Ta obvestila se berejo iz datotek s prevodi, tako da lahko naredimo aplikacijo večjezično. Ogrodje vedno proži napako tipa *AppException* ali kakšno izmed njenih naslednikov. Posebnost teh napak je, da kot prvi parameter vedno prejmejo šifro napake, prevod pa se prebere iz datoteke z opisi vseh možnih napak, ki se nahaja v mapi messages, imenuje pa se *errorMessages.i18n*. Opisi teh napak naj bi bili praviloma v angleščini, saj je dobra praksa, da se v dnevnik napak zapisujejo angleška sporočila. Vsakemu opisu napake, lahko definiramo še dodaten ID prevoda, ki se nahaja v datotekah s prevodi. Nazivi teh datotek so sestavljeni po

naslednjem vzorcu *messages.<oznaka jezika>.i18n*. Datoteka s slovenskimi prevodi se tako imenuje *messages.sl.i18n*, datoteka z angleškimi prevodi pa *messages.en.i18n*, itd.

Ogrodje ob zagonu aplikacije avtomatsko prebere vse datoteke s prevodi, ki jih lahko potem uporabi za izpis napak. V teh datotekah ni pomembno, da se nahajajo prevodi vseh sporočil o napakah, ampak naj bi se prevedle le tiste, ki so za končnega uporabnika aplikacije pomembne, saj se pri vsakem sporočilu o napaki lahko, ali pa ne, definira sporočilo iz datotek s prevodi.

Končnemu uporabniku praviloma ne prikazujemo vsakega opisa napake, ki se zgodi, saj mu verjetno tudi nič ne bo pomenila. Na tem mestu pride do izraza mehanizem za napake, saj lahko originalno sporočilo napake zapišemo v dnevnik napak, prevod pa pošljemo na uporabniški vmesnik. Več različnih napak ima lahko enak prevod za uporabnika, ki je bolj splošne narave. Pomembno je le, da imamo v dnevniku vedno zapis o dejanski napaki, ki se je zgodila.

Vse datoteke s prevodi in glavna datoteka z opisi napak so napisani kot skripti jezika Groovy. Ogrodje ob zagonu aplikacije tudi preveri ali so datoteke s prevodi usklajene med seboj. To stori tako, da preveri ali v kateri datoteki kakšen izmed prevodov, ki obstaja v ostalih, manjka. V takem primeru sproži napako in aplikacija se ne zažene, dokler ne odpravimo napak v prevodih. Le na tak način lahko zagotovimo, da so datoteke s prevodi vedno usklajene in da ne bo prihajalo do napak med branjem določenega prevoda.

4.3 Arhitektura

Naše aplikacijsko ogrodje je napisano v programskem jeziku Java. Za razvoj ogrodja nismo uporabili nobenega obstoječega ogrodja, ki služi hitremu razvoju aplikacij, kot so na primer Spring, Dropwizard, Spark in podobni, ampak je napisano le z uporabo razredov, ki so del standardnega javanskega razvojnega paketa Java Development Kit (JDK). Uporabimo ga lahko tako za razvoj spletnih aplikacij, ki tečejo na aplikacijskih strežnikih, za samostojne oziroma namizne aplikacije, ki tečejo na odjemalcu, lahko se z njegovo pomočjo razvija mikrostoritve, ki so dandanes zelo popularen način razvoja aplikacij, itd.

Ogrodje vsebuje razred *Model*, ki predstavlja enoten API za dostop do podatkov iz podatkovnih virov in ni pomembno, kakšno vrsto vira uporablja aplikacija. Zaradi takšnega načina dostopa do podatkov ni potrebno spreminjati programske kode aplikacije, če želimo aplikacijo premestiti na drug podatkovni vir. Razred Model je tesno povezan z razredom *RuleEngine*, ki poskrbi za izvajanje vseh poslovnih pravil posamezne entitete pred in po komunikaciji s podatkovnim virom. Model na podlagi vrste entitete sam ugotovi v kateri podatkovni vir sodi določen zapis, tako da se razvijalcu aplikacije s tem ni potrebno ukvarjati. To mu omogoči, da

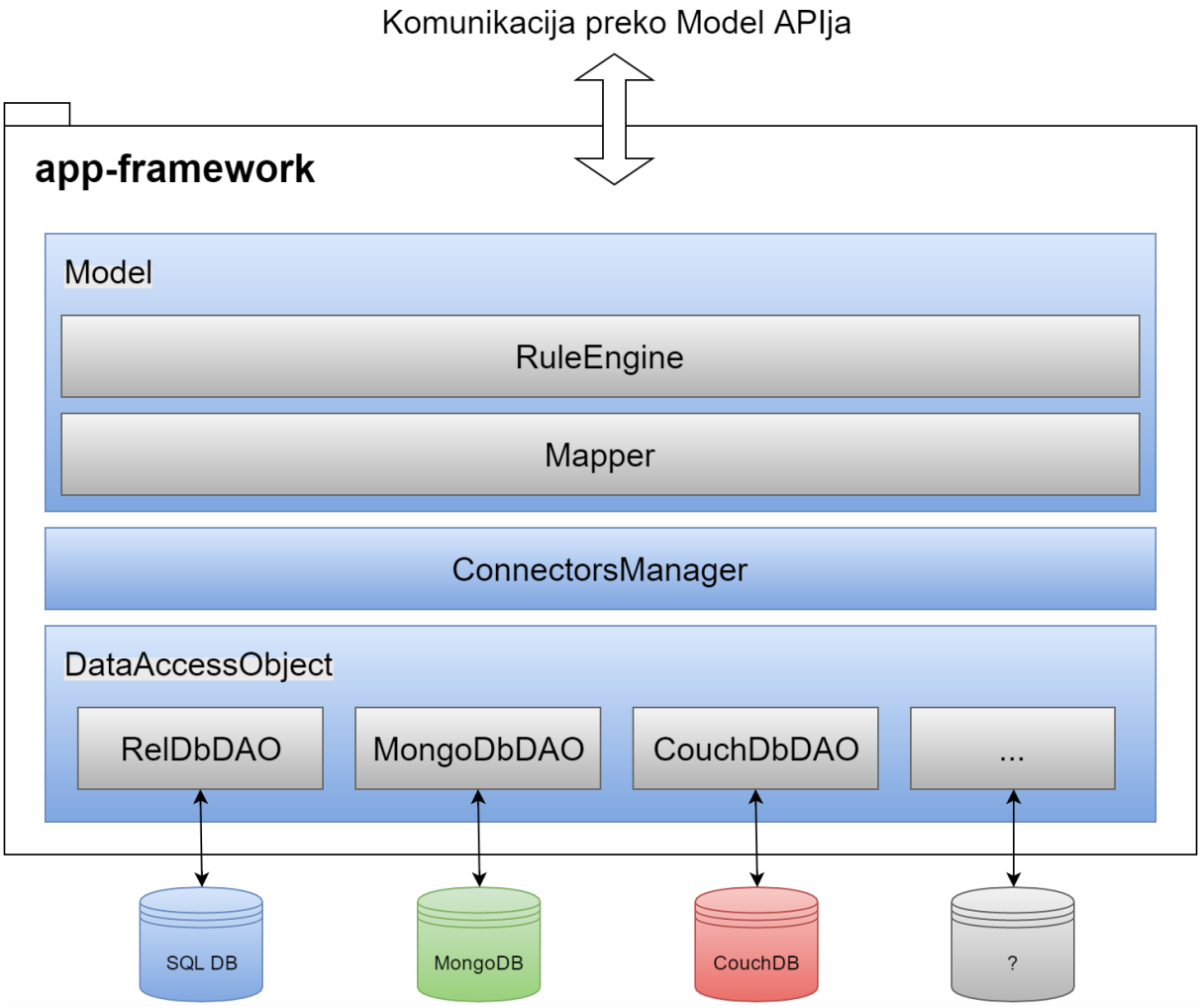
se bolj osredotoči na pisanje same poslovne logike, kot pa na to, kam in kako bo podatke zapisal, oziroma od kje jih bo bral. Razvijalcem tako tudi ni potrebno poznati načina dela z določenim podatkovnim virom, kot na primer, kako vzpostaviti povezavo do podatkovnega vira, kako pravilno sproščati zasežene vire (pravilno zapiranje povezav ter podatkovnih tokov, ko jih ne potrebujemo več), na kakšen način je potrebno definirati poizvedbe za vse CRUD operacije, ipd. To je lahko velik plus, saj tako v podjetju ni potrebno imeti specialistov, ki dobro poznajo načine dostopa za izbran podatkovni vir, saj za dostop do podatkov skrbi ogrodje samo, oziroma razredi, ki so napisani za določeno vrsto podatkovnega vira.

Človeka, ki se spozna na delo z določenim tipom podatkovnega vira, potrebujemo le takrat, ko želimo ogrodje razširiti z novim razredom za komunikacijo z želenim podatkovnim virom. To je navadno bolj izkušen razvijalec, saj mora biti komunikacija s podatkovnim virom narejena karseda učinkovito. Poleg tega razvijalci začetniki velikokrat ne poskrbijo za pravilno sproščanje zaseženih virov, kar v nadaljevanju vodi do napake, ki se imenuje puščanje pomnilnika (to pomeni, da aplikacija sčasoma porabi vse vire in se neha odzivati, oziroma pride do napake, ki sporoča, da je zmanjkalo prostega pomnilnika). Najpogostejše napake tega tipa pri začetnikih so maksimalno število odprtih povezav na podatkovno bazo (ker se jih ustrezno ne zapira, ko le-te niso več potrebne), poln pomnilnik (če ni ustreznega sproščanja podatkovnih tokov), nezaprte povezave na podatkovni vir v primeru napake (neustrezna obravnava izjem) itd.

Slika 4.1 prikazuje poenostavljeno arhitekturo najpomembnejših razredov našega ogrodja. Sliko beremo od zgoraj navzdol, kar pomeni, da je na vrhu razred Model, ki sprejme zahtevek za dostop do podatkov. Razred Model implementira javne funkcije, oziroma izpostavlja API našega ogrodja. Vsebuje vse funkcije, ki jih lahko uporabimo za zahtevke, ki upravljajo s podatki (CRUD). Razred Model za ustrezno izvajanje poslovnih pravil uporablja funkcionalnosti razreda RuleEngine. Tam se nahaja vsa logika, ki proži ustrezne funkcije posamezne entitete napisane za določeno CRUD operacijo. Poleg razreda RuleEngine Model uporablja tudi razred Mapper, ki poskrbi za ustrezno preslikovanje objektov, če imamo v aplikaciji definiran preslikovalnik za izbrano entiteto. Če je zahtevek prestal vse preslikave ter poslovna pravila pomeni, da gre lahko naprej, kjer se dejansko sproži branje oziroma pisanje v podatkovni vir. Za ta namen je razvit razred ConnectorsManager, ki vsebuje logiko, da za trenutni objekt na podlagi atributa »entity« določi, v kateri podatkovni vir sodi. ConnectorsManager te podatke pridobi ob zagonu aplikacije, ko se prebere nastavitvena datoteka, kjer so definicije vseh podatkovnih virov s seznami entitet, ki sodijo vanje. Za vsak tip podatkovnega vira je potrebno razviti razred, ki ga zna uporabljati. Vsak tak razred mora implementirati vmesnik (angl. interface) DataAccessObject, ki vsebuje vse potrebne CRUD

operacije. Kot je razvidno s slike 4.1, smo že v okviru diplomske naloge razvili tri različne objekte, ki znajo komunicirati z različnimi podatkovnimi bazami:

- RelDbDAO je razred, ki zna preko protokola JDBC komunicirati z vsako relacijsko bazo, ki implementira poizvedovalni jezik SQL. Ob inicializaciji potrebuje le podatek o razredu, ki za ciljno podatkovno bazo implementira funkcije protokola JDBC (JDBC gonilnik, oziroma angl. JDBC driver);
- MongoDbDAO je razred, ki implementira CRUD operacije za delo s podatkovno bazo MongoDb;
- CouchDbDAO pa razred za komunikacijo s podatkovno bazo CouchDb.



Slika 4.1: Arhitektura razredov aplikacijskega ogrodja

Če želimo uporabiti podatkovni vir, ki ga ogrodje še ne podpira, je potrebno le razviti nov razred tipa DataAccesssObject za želeno podatkovno bazo oziroma podatkovni vir in že ga lahko uporabimo v nastavitvah ogrodja za želeno entiteto.

4.4 Kako razviti aplikacijo s pomočjo našega aplikacijskega ogrodja

Za potrebe ponazoritve razvoja aplikacije s pomočjo našega ogrodja smo razvili tudi preprosto testno aplikacijo app-framework-tester, ki vsebuje JUnit teste za osnovne CRUD (Create, Read, Update, Delete) operacije nad različnimi podatkovnimi bazami. Testna aplikacija služi tudi za zgled, da lahko ogrodje učinkovito uporabimo za testiranje učinkovitosti različnih podatkovnih virov. V teste smo vključili štiri podatkovne baze:

1. **CouchDB 1.6.1** – dokumentna NoSQL podatkovna baza;
2. **MongoDB 3.2.1** - dokumentna NoSQL podatkovna baza;
3. Različica zelo popularne in razširjene relacijske podatkovne baze MySQL – **MariaDB 10.1.9**;
4. ter **PostgreSQL 9.5** – relacijska podatkovna baza.

Kot je razvidno, ogrodje ni omejeno na eno vrsto podatkovnih baz, ampak je razvito na način, da lahko hitro napišemo nov razred, ki implementira vmesnik *DataAccesObject*, ki vsebuje vse osnovne CRUD operacije, ki jih lahko potem izvajamo nad podatki. Preko takega razreda potem poteka vsa komunikacija za določen tip podatkovnega vira.

4.4.1 Konfiguracija

Za uspešen zagon aplikacije, ki jo razvijamo s pomočjo našega ogrodja, se mora v glavni mapi projekta nahajati nastavitvena datoteka *app.config*, ki se interpretira s pomočjo programskega jezika Groovy. Nastavitvena datoteka mora definirati naslednje vrednosti, ki so potrebne za delovanje aplikacije:

- *modelPackage* – ime paketa, kjer bodo definirani razredi, ki predstavljajo posamezne entitete;
- *rulesPackage* – ime paketa, kjer bodo definirani razredi s poslovnimi pravili, ki se izvajajo pred ali po komunikaciji s podatkovnim virom;
- *entities* – seznam vseh entitet, ki nastopajo v aplikaciji;
- seznam definicij za dostop do vseh podatkovnih virov, ki jih uporablja aplikacija.

Na sliki 4.2 lahko vidimo primer nastavitvene datoteke *app.config*, ki smo jo uporabili v naši testni aplikaciji. Kot je razvidno, smo uporabili štiri entitete; PERSON, postgre_person, mysql_person in mongodb_person. Vsaka izmed njih gre v svoj podatkovni vir, kot je definirano s kodnimi bloki, ki se pošljejo v funkcijo *dataSource*. Pri glavni definiciji podatkovnega vira (vedno je definirana prva) ne definiramo seznama entitet, ki spadajo vanj, saj je to definirano implicitno (vanj spadajo vse entitete, ki niso uporabljene v sekundarnih podatkovnih virih). V sekundarnih podatkovnih virih pa lahko definiramo vsebovane entitete tako, da jih dodamo v seznam *entityList* (glej sliko 4.2).

```
modelPackage = 'si.fri.vonta.sampleApp.model'
rulesPackage = 'si.fri.vonta.sampleApp.rules'

entities = [
      'PERSON',
      'postgre_person',
      'mysql_person',
      'mongodb_person'
]

/*
 * Definicija privzetega podatkovnega vira se uporablja za vse entitete,
 * ki niso definirane v kakšnem izmed ostalih definicij.
 */
dataSource {
      type = 'couchDb'
      host = 'localhost'
      port = 5984
      dbName = 'couchdb'
}

dataSource 'PostGreSQL', {
      type = 'relDb'
      driver = 'org.postgresql.Driver'
      connectionString = 'jdbc:postgresql://192.168.1.201:5432/persons'
      username = 'postgres'
      password = 'admin'

      entityList << 'postgre_person'
}

dataSource 'MySQL', {
      type = 'relDb'
      driver = 'com.mysql.jdbc.Driver'
      connectionString = 'jdbc:mysql://192.168.1.201/employees'
      username = 'user'
      password = 'user'

      entityList << 'mysql_person'
}

dataSource 'MongoDB', {
      type = 'mongoDb'
      host = '192.168.1.201'
      dbName = 'local'

      entityList << 'mongodb_person'
}
```

Slika 4.2 Primer nastavitvene datoteke *app.config*

4.4.2 Datoteke z opisi entitet

Datoteke z opisi entitet, ki nastopajo v aplikaciji, se morajo nahajati v mapi *def*. Tudi te datoteke so v osnovi skriptne datoteke jezika Groovy. Možni sta dve vrsti datotek, in sicer *.descriptor

in *.mapper. Za naziv datoteke je obvezno potrebno navesti ime entitete, ki je bila definirana v nastavitveni datoteki *app.config* v seznamu entitet *entities*.

Datoteke tipa *descriptor* vsebujejo naslednje definicije oz. ukaze:

- extend – ukaz extend pove, katero drugo entiteto razširja definirana entiteta (s tem se izognemo ponavljajočem in redundantnem definiranju atributov entitet);
- idField – naziv unikatnega identifikatorja zapisa v bazi;
- revField (neobvezno – definiramo le, če ga entiteta uporablja) – naziv polja, ki se uporablja za ugotavljanje konfliktov ob shranjevanju iz uporabniškega vmesnika pri večuporabniških aplikacijah;
- fields – asociativno polje z naborom vseh polj, ki jih vsebuje entiteta. Uporablja se tudi za izgradnjo privzetega preslikovalnika, ki se uporabi pred zapisom v podatkovni vir. To pomeni, da če nekega atributa entitete tukaj ne definiramo, se tudi v podatkovni vir ne bo zapisal;
- views – seznam vseh možnih indeksov po katerih se lahko izvaja poizvedbe. V primeru relacijskih podatkovnih baz, se tukaj definira seznam indeksiranih stolpcev entitete, v primeru CouchDBja se tukaj definira naziv pogleda oziroma viewja.

Pri vsaki definiciji polja lahko definiramo tudi nekaj dodatnih lastnosti. Lahko definiramo, da je neko polje obvezno, da mora biti ID neke druge entitete, itd. Vse te dodatne lastnosti in pravila se preverijo pred vsakim zapisom, pa naj gre za novo kreiranje objekta ali za posodobitev obstoječega zapisa. Če katerikoli atribut ne zadosti definiranim pravilom, se proži napaka in niti ne pride do interakcije s podatkovnim virom.

Slika 4.3 prikazuje primer opisne datoteke za neko entiteto (to je dejanski opis entitete mysql_person iz naše testne aplikacije). Kot je razvidno iz primera, je primarni ključ zapisa osebe v podatkovni bazi v stolpcu *id*. V polju *fields* so navedeni vsi stolpci v tabeli oziroma vse lastnosti entitete (lahko definiramo tudi obveznost lastnosti). Pozoren bralec bo opazil, da lastnost *id* ni obvezna, čeprav vemo, da primarni ključ v relacijski bazi ne more biti prazna vrednost. To seveda ni napaka, ampak je tako definirano namenoma. Ko pri neki lastnosti definiramo, da je obvezna, je to potrebno gledati iz stališča našega ogrodja. Ogrodje preveri prisotnost lastnosti entitete še preden jo kreira ali posodobi v podatkovnem viru. Naša testna aplikacija je bila zasnovana tako, da so vse baze primarni ključ ustvarile samodejno ob kreiranju novega zapisa. Torej to pomeni, da bi ob kreiranju novega zapisa ogrodje sprožilo napako še preden bi prišlo do dejanskega zapisa v bazo, ker lastnost *id* ne bi imela vrednosti. Polje *views* pa definira filter z nazivom *postnumber*, ki vsebuje le eno istoimensko lastnost. To smo storili zato, da bo ogrodje prožilo iskanje v podatkovnih bazah z dodanim pogojem, ki predstavlja

poštno številko. Testna aplikacija je bila načrtovana tako, da so vse uporabljene podatkovne baze imele indeksirano lastnost *postnumber*.

```
idField = 'id'

fields = [
      id:          [:],
      name:        [required: true],
      surname:     [required: true],
      email:       [:],
      street:      [:],
      postnumber:  [:],
      postname:    [:],
      birthdate:   [required: true],
      emso:        [required: true],
]

views = [
      postnumber: ['postnumber']
]
```

Slika 4.3 Primer datoteke z opisom entitete **.descriptor*

Datoteke tipa *mapper* vsebujejo definicije osnovnih preslikovalnikov za posamezno entiteto. Definiramo jih enako, kot so definirana vsa možna polja določene entitete v polju *fields*.

4.4.3 Datoteke, ki predstavljajo objekte entitete

Vsaka entiteta, ki jo definiramo v seznamu entitet v nastavitveni datoteki app.config, mora imeti pripadajoči razred z enakim nazivom, ki razširja razred Entity. Tako imamo za vsako entiteto njen pripadajoči razred, ki se uporablja za lažji dostop do njenih atributov, saj imamo po navadi definirane tako imenovane GET in SET metode. Razred Entity je podrazred razreda Map, ki predstavlja asociativna polja, ki so najbližje JSON objektu, ki služijo za REST komunikacijo. Ker je naše ogrodje mešanica javanske kode in Groovy kode, je to tudi najbolj primeren objekt za entitete, saj se z Map objekti v Groovyju zelo elegantno upravlja z osnovnimi ukazi, oziroma je sintaksa za Map objekte sila preprosta.

4.4.4 Datoteke s poslovnimi pravili

Vsaka entiteta ima lahko definiran razred s specifičnimi poslovnimi pravili za dotično entiteto. Če tega razreda ni, se uporabi privzeti razred, ki predstavlja razred s pravili – DefaultRules. Razred s poslovnimi pravili definiramo tako, da se naziv datoteke začne z imenom entitete in nadaljuje z »_Rules.java«, ter mora obvezno razširjati DefaultRules, ki je privzeti razred s pravili.

Vse CRUD operacije imajo »pred« in »po« pravila. Če na primer želimo kreirati nov zapis določene entitete, se pred dejanskim kreiranjem izvedejo »pred« pravila za dotično entiteto in če napak ni bilo, sledi dejanski zapis v podatkovni vir. Po uspešnem vpisu objekta v podatkovni vir sledi izvajanje »po« pravila. Tak način nam omogoča fleksibilnost pri definiranju pravil za vsako entiteto posebej.

Za vsako izmed entitet se ob zagonu aplikacije tvori tudi osnovni preslikovalnik, kot smo to omenili že v prejšnjih poglavjih. Ta preslikovalnik se zgradi iz seznama polj v opisni datoteki entitete. RuleEngine poleg izvajanja »pred« in »po« pravil definiranih za vsako entiteto posebej poskrbi tudi za potrjevanje entitete z osnovnim preslikovalnikom pred vsakim kreiranjem ali posodobitvijo v podatkovnem viru. Tako imamo zagotovljeno veljavno stanje vsakega zapisa, saj v nasprotnem primeru pride do napake in zapis ne doseže podatkovnega vira. Osnovni preslikovalnik odstrani vse morebitne dodatne atribute, ki niso navedeni v opisni datoteki entitete. To pri relacijskih podatkovnih bazah niti ne pride tako do izraza, saj imamo tam točno definirano podatkovno shemo, tako da v določeno tabelo ne moremo shraniti poljubnih atributov, ampak imamo točno določene stolpce. Pri dokumentnih podatkovnih bazah, pa to pride še kako prav, saj tam podatkovna shema ni definirana in če ne bi odstranili nepotrebnih atributov, bi se ti zapisali v bazo. To s tehničnega vidika seveda ne bi bil problem, vendar to vodi do slabše berljivosti podatkov in zasedanja dodatnega pomnilniškega prostora, saj bi imel vsak zapis poleg potrebnih atributov še kopico takih, ki nimajo semantične vrednosti za aplikacijo. Preslikovalnik preveri tudi obveznost in veljavnost atributov.

4.4.5 Zagon aplikacije

Instanco aplikacije lahko zaženemo, ko so ustrezno kreirane vse potrebne opisne datoteke, ki smo jih opisali v prejšnjih poglavjih. To storimo tako, da pokličemo funkcijo start(), ki se nahaja v razredu Application. Ob zagonu aplikacije se najprej prebere nastavitvena datoteka. Vzpostavijo se vse povezave, ki so definirane v nastavitvah.

Takoj za tem se preverijo vse definicije za vsako izmed entitet, kreirajo se vsi preslikovalniki polj, ki se uporabljajo ob branju in zapisovanju entitet v podatkovne vire, preveri se tudi obstoj in pravilnost razredov, ki predstavljajo posamezno entiteto. Če je vse v redu, se aplikacija uspešno zažene in je pripravljena za uporabo.

Za delovanje ogrodja je nujen zagon aplikacije preko start() metode v razredu Application. Iz tega razloga postavimo klic Application.start() na ustrezno mesto, odvisno od tipa aplikacije, ki uporablja naše ogrodje.

Če ogrodje uporabimo pri namizni aplikaciji, bo zagon aplikacije verjetno nekje na začetku main metode. Za uporabo recimo v spletni aplikaciji, bo klic za zagon aplikacije v servletu, ki se prvi zažene ob zagonu spletne aplikacije. Pri testiranju programskih enot bo zagon aplikacije v metodi, ki se zažene pred vsemi testi – torej v tisti metodi, ki je definirana z oznako @BeforeClass.

Razred Application ima poleg start() metode tudi metodo stop(), ki poskrbi za ustrezno ustavitev aplikacijskega ogrodja, tako da ustrezno zapre vse povezave na podatkovne vire in tako sprosti zasežene vire. Aplikacija mora biti napisana tako, da ob zaključku vedno poskrbi, da se pokliče metoda stop().

4.5 Predstavitev testne aplikacije

Za ponazoritev uporabe našega testnega aplikacijskega ogrodja smo naredili manjšo testno aplikacijo, ki služi kot primerjava različnih podatkovnih baz. Ker je eden izmed namenov ogrodja enoten API za dostop do podatkov, smo za vsak test napisali eno funkcijo, ki testira določen del dostopa do podatkov, potem pa to uporabili za vsako izmed podatkovnih baz. Tako se za vsak test uporabi popolnoma identična programska koda, vendar se uporabi druga podatkovna baza za shrambo podatkov.

Za testiranje smo uporabili orodje JUnit verzije 4.12. Testirali smo štiri osnovne operacije podatkovnih baz:

- InsertDbTest - vnos podatkov v podatkovno bazo;
- findDbTest – iskanje zapisov, katerih atribut ustreza podani vrednosti;
- updateDbTest – posodobitev zapisa;
- deleteDbTest – brisanje zapisa iz podatkovne baze.

V okviru testiranja smo uporabili štiri podatkovne baze, ki so bile vse nameščene na enaki strojni in programski opremi, tako da so rezultati med seboj čim bolj primerljivi. Dve izmed podatkovnih baz sta bili dokumentni oziroma NoSQL podatkovni bazi, to sta CouchDB 1.6.1 in MongoDB 3.2.1. Drugi dve podatkovni bazi pa sta bili relacijski, in sicer MariaDB 10.1.9 ter PostgreSQL 9.5. Vsi testi so napisani tako, da zaporedno izvajajo operacije nad določeno bazo. Test za vnos najprej vnese vse zapise v eno bazo, potem vse zapise v drugo, itd. Vsakega izmed testov smo pognali večkrat, tako da smo na koncu lahko izračunali povprečne vrednosti rezultatov. V naslednjih poglavjih bomo prikazali povprečne rezultate testov za posamezne CRUD operacije.

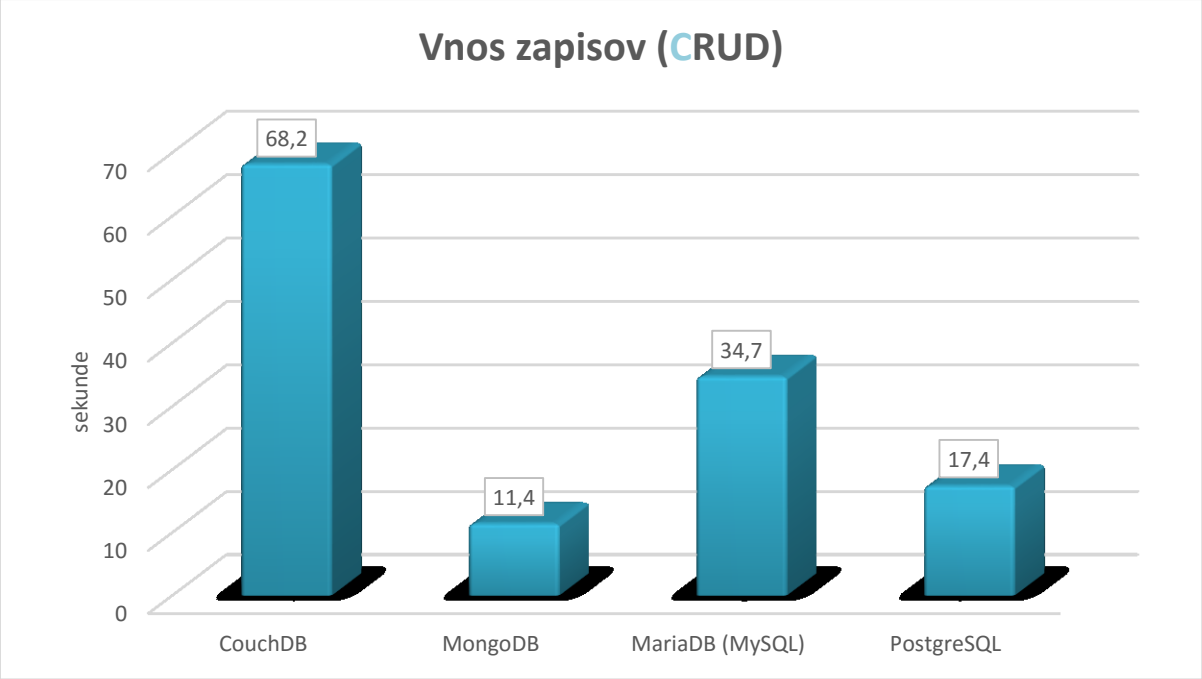
Testna aplikacija je bila načrtovana tako, da je vsaka izmed podatkovnih baz imela le eno entiteto, kar pomeni le eno tabelo v relacijskih podatkovnih bazah, eno kolekcijo dokumentov v MongoDB podatkovni bazi ter eno entiteto v CouchDB podatkovni bazi. Vsi zapisi v podatkovnih bazah so bili naključno generirani podatki o osebah. Vsaka izmišljena oseba je bila sestavljena iz naslednjih podatkov:

- ime – naključen niz dolžine od 3 do 10 znakov;
- priimek – naključen niz dolžine od 5 do 20 znakov;
- e-pošta – niz sestavljen kot ime.priimek@test.com;
- EMŠO – naključen niz 13-ih števk;
- datum rojstva – naključen datum rojstva med datumom 1.1.1930 in 31.12.2009;
- ulica in hišna številka – naključen niz sestavljen iz dveh naključnih nizov dolžine med 5 in 12 znakov ter hišne številke med 1 in 999;
- poštna številka – naključna številka med 1000 in 1999;
- pošta – naključen niz dolžine od 5 do 25 znakov.

Za unikaten ID (identifikator) je bila uporabljena zaporedna številka vnosa zapisa v bazo, ki je bila pri vseh štirih podatkovnih bazah tipa niz. V relacijski bazi je lahko primarni ključ številskega ali znakovnega tipa, CouchDB pa zahteva, da je ID vedno znakovnega polja, zato smo se odločili, da bo ID v vseh štirih bazah znakovni niz.

4.5.1 insertDbTest – vnos zapisov v baze

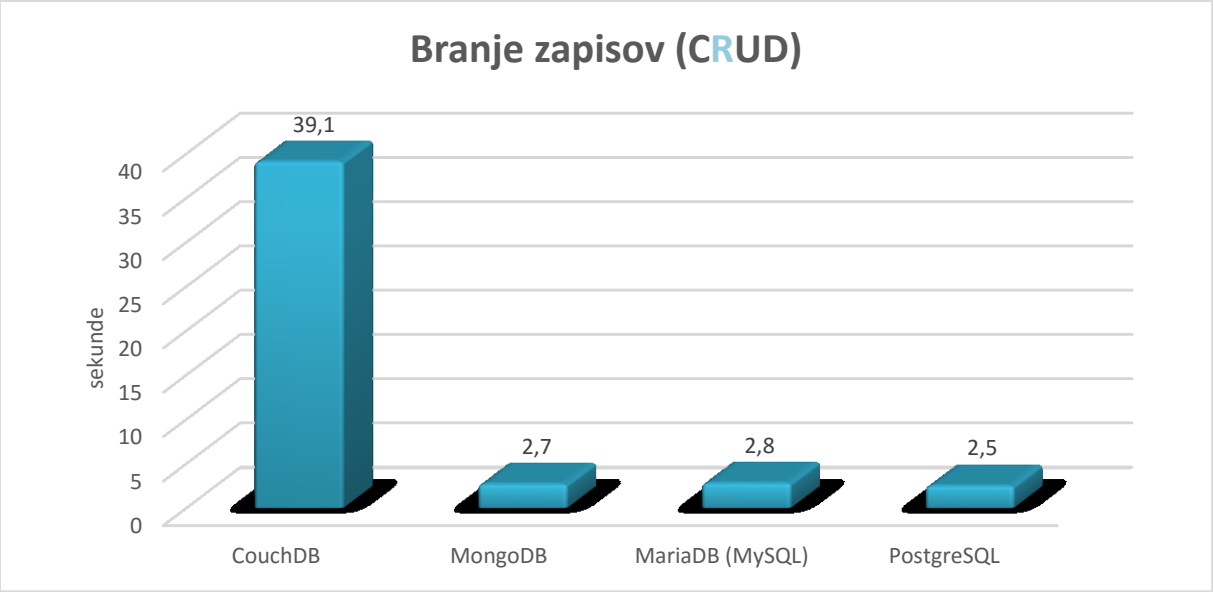
Prvi izmed testov, ki smo ga pognali, je bil test, ki je prazno podatkovno bazo napolnil. Test je vseboval zanko, ki je v vsako izmed podatkovnih baz vnesel 30.000 zapisov, ki so bili naključno generirani podatki oseb, kot smo to opisali v poglavju 4.5. Slika 4.4 prikazuje čas v sekundah, ki ga je vsaka izmed podatkovnih baz potrebovala za vnos 30.000 naključnih oseb.



Slika 4.4 Porabljen čas za vnos 30.000 zapisov

4.5.2 findDbTest – iskanje zapisov

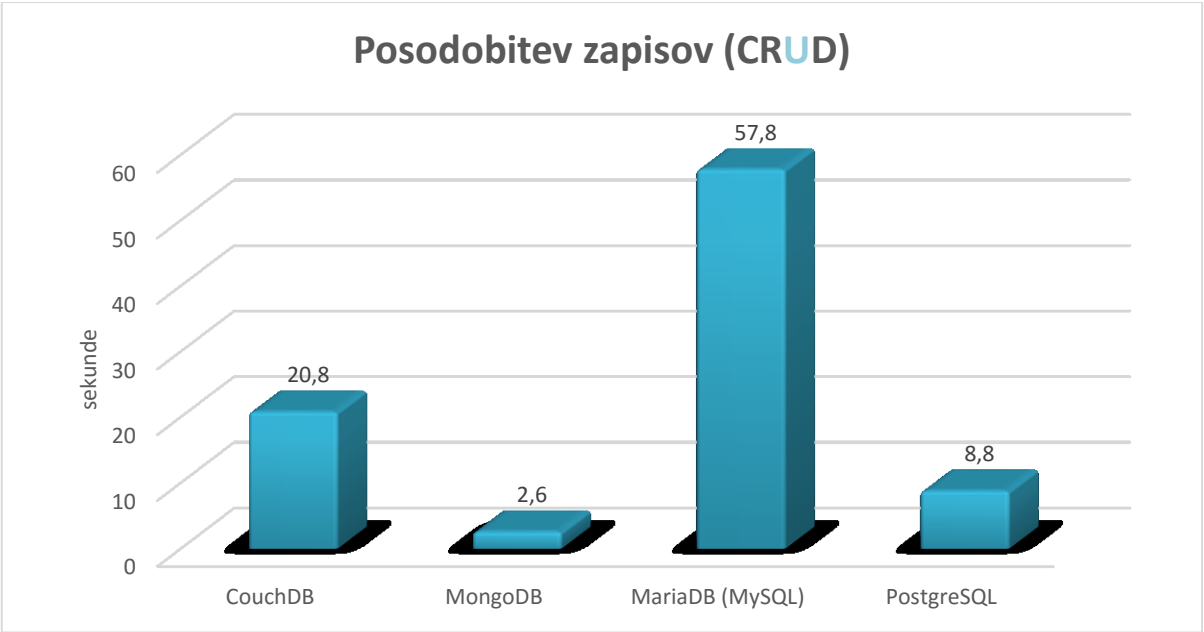
Drugi test, ki smo ga poganjali, je bil test za iskanje zapisov. Vsa iskanja so bila izvedena na podlagi atributa poštna številka, ki je bil v vseh štirih bazah indeksiran. To pomeni, da je iskanje po takem atributu najhitreje, kar zmore posamezna baza. Test je bil napisan tako, da je nad vsako bazo sprožil natanko 3.000 iskalnih poizvedb z naključno generirano poštno številko. Slika 4.5 prikazuje čas v sekundah, ki ga je vsaka izmed podatkovnih baz potrebovala za 3.000 iskanj.



Slika 4.5 Porabljen čas za izvedbo 3.000 poizvedb

4.5.3 updateDbTest – posodobitev zapisov

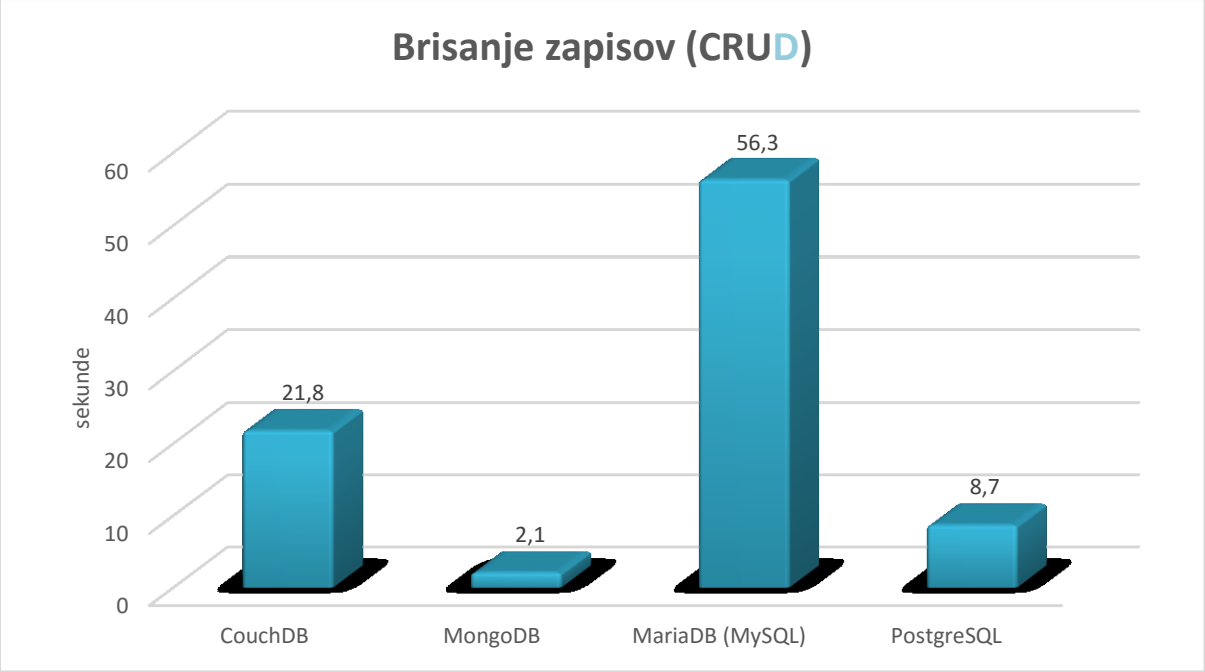
Tretji test, ki smo ga poganjali nad testnimi bazami, je bil test posodobitve. Test je bil napisan tako, da prebere naključni zapis iz podatkovne baze, mu generira nov naključen priimek ter ga posodobi v bazi. Za vsako bazo se je omenjena procedura izvedla natanko 1000-krat. Slika 4.6 prikazuje čas v sekundah, ki ga je vsaka izmed podatkovnih baz potrebovala za posodobitev naključnih 1.000 oseb.



Slika 4.6 Porabljen čas za posodobitev 1.000 zapisov

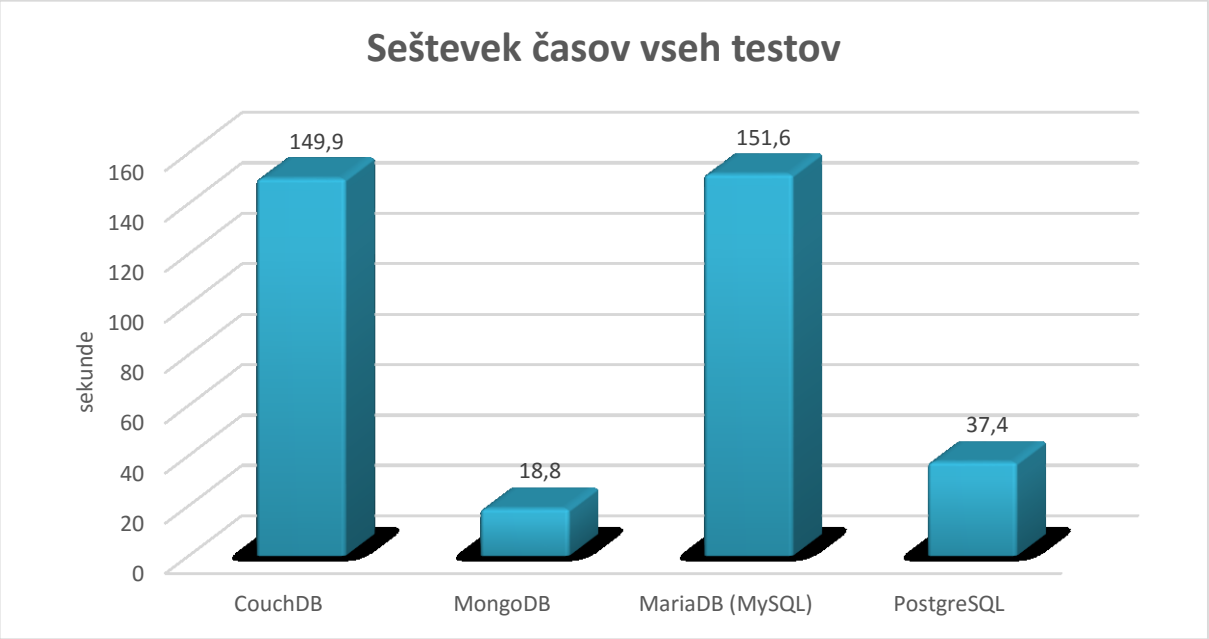
4.5.4 deleteDbTest – brisanje zapisov

Zadnji izmed testov, ki so testirali CRUD operacije nad podatkovnimi bazami, je bil test brisanja zapisov. Test je v vsaki izmed baz pobrisal 1.000 naključnih zapisov. Slika 4.7 prikazuje čas v sekundah, ki ga je vsaka izmed podatkovnih baz potrebovala za izbris 1.000 naključnih oseb.



Slika 4.7 Porabljen čas za izbris 1.000 zapisov

4.5.5 Analiza rezultatov



Slika 4.8 Seštevek časov vseh testov

Slika 4.8 prikazuje seštevke porabljenega časa vseh štirih testov za vsako podatkovno bazo posebej. Glede na rezultate naših preprostih CRUD testov se je daleč najbolje izkazala NoSQL podatkovna baza MongoDB, ki je vse štiri teste opravila v manj kot 19-ih sekundah. Druga najhitrejša baza PostgreSQL je za izvedbo vseh testov potrebovala skoraj dvakrat več časa, kar znese dobrih 34 sekund. Ostali dve podatkovni bazi, CouchDB in različica MySQL baze MariaDB, sta obe potrebovali skoraj 2 minuti in pol, kar je velika razlika v primerjavi s prvima dvema.

Pomemben razlog za tako dobre rezultate podatkovne baze MongoDB izhaja iz tega, da ne podpira transakcij [7], kar zelo poenostavi algoritem, ki skrbi za CRUD operacije. To tudi pomeni, da ni skladna z modelom ACID. ACID je angleška kratica za štiri koncepte, katerim mora zadostiti vsaka transakcija:

- Atomarnost (angl. atomicity) – ali se izvedejo vsa opravila določene transakcije ali pa nobeno;
- konsistentnost (angl. consistency) – transakcija mora zagotoviti, da se v podatkovno bazo zapišejo le podatki, ki ustrezajo vsem pravilom za zagotavljanje konsistence med podatki;

- izolacija (angl. isolation) – zagotavlja, da kljub vzporednemu izvajanju več transakcij ne pride do primera, ko bi neka transakcija povozila podatke neki drugi transakciji;

- Trajnost (angl. durability) – uspešna potrditev transakcije (angl. commit) zagotavlja, da so podatki zagotovo zapisani v bazo in ne more priti do izgube le-teh.

Podatkovna baza CouchDB prav tako ni skladna z modelom ACID, vendar pa je razlog za slabe rezultate sama komunikacija. Za komunikacijo uporablja protokol HTTP, ki je precej počasnejši od binarnega protokola JDBC, ki se ga uporablja pri ostalih treh podatkovnih bazah. Protokol HTTP porabi veliko dodatnega časa za samo vzpostavitev povezave. Relacijski podatkovni bazi MariaDB in PostgreSQL pa sta obe skladni z modelom ACID [13]. Vendar kot kažejo naši rezultati je PostgreSQL učinkovitejša od baze MariaDB, ki se je zelo slabo izkazala ob posodobitvah in brisanju podatkov.

Za bolj natančno primerjavo podatkovnih baz bi potrebovali več različnih testov, ki bi na različne načine prožili poizvedbe. Potrebovali bi še teste, ki naključno prožijo vse tipe zahtevkov (kreiranje, branje, posodabljanje in brisanje), testirali bi lahko tudi, kako se baze obnašajo pri zelo velikih količinah podatkov (tabele z nekaj milijonov zapisov) in še mnogo drugih testov.

Vendar pa namen naše diplomske naloge ni bila primerjava podatkovnih baz, ampak je bil to le primer, ki ponazarja, kako lahko z našim aplikacijskim ogrodjem hitro in z zelo malo programske kode izdelamo aplikacijo, ki zna brati podatke iz različnih podatkovnih virov, pri tem pa uporablja enoten API.

Poglavje 5 Sklepne ugotovitve

V okviru diplomske naloge je bilo uspešno razvito aplikacijsko ogrodje za hiter razvoj aplikacij. Ogrodje omogoča definiranje več različnih podatkovnih virov ter vseh entitet, ki nastopajo v aplikaciji. V nastavitveni datoteki je vedno definiran en privzeti podatkovni vir, kamor spadajo vse entitete. Kadar imamo opravka z več entitetami, ki se nahajajo na različnih podatkovnih virih, v nastavitvah definiramo dodatne podatkovne vire s seznami entitet, ki se tam nahajajo. Aplikacija na ta način ve, katera entiteta spada v kateri podatkovni vir. Kot smo definirali v zahtevah na začetku, je izdelan enoten API za dostopanje do podatkov, ne glede na vrsto podatkovnega vira, kar poenostavi uporabo ogrodja, ter omogoča popolno prenosljivost aplikacije na drug podatkovni vir brez sprememb v kodi. Poleg tega nam ponuja tudi preprost način definiranja poslovnih pravil za vsako izmed entitet. Pravila se delijo na »pred« in »po« pravila, kar pomeni, da se izvedejo pred ali po določeni CRUD operaciji.

Poleg prej omenjenih glavnih funkcionalnosti je bilo razvitih še precej drugih, ki pripomorejo k uporabnosti in učinkovitosti ogrodja. Ena izmed takih je dvonivojski izpis napak, ki omogoča izpis originalne napake v dnevnik napak, oziroma na standardni izhod, obvestilo za uporabnika pa se lahko razlikuje od originalnega opisa napake. Tak način je prijazen tako uporabniku, kot tudi razvijalcem programskih rešitev, saj omogoča izpis uporabniku prijaznega sporočila, hkrati pa ima razvijalec zabeleženo originalno napako, ki mu pomaga odkriti težavo, zaradi katere se je sprožila izpisana napaka. Uporabniški izpisi napak so definirani v jezikovnih datotekah, tako da lahko na preprost način ogrodje razširimo z novim jezikom na način, da naredimo le kopijo obstoječe jezikovne datoteke in v ime zapišemo dvočrkovno oznako novega jezika, ter seveda prevedemo sporočila. Poleg tega je izdelan mehanizem za definiranje preslikovalnikov na nivoju entitet, ki omogoča, da določeni entiteti dodamo kakšen izpeljan atribut ali izpustimo enega ali več atributov, ki jih vsebuje v podatkovnem viru. To je še posebej uporabno, kadar razvijamo aplikacijo, ki integrira podatke iz različnih že obstoječih podatkovnih virov, vendar ne potrebujemo vseh atributov izbranih entitet.

Ogrodje je zasnovano tako, da ga lahko na preprost način nadgradimo z dodatnimi funkcionalnostmi. Z uporabo načrtovalskega vzorca Command pattern smo zagotovili, da lahko z res preprosto nadgraditvijo ogrodja omogočimo razvijanje večuporabniških aplikacij. Razred Model, ki je s svojim APIjem vstopna točka v ogrodje, definira funkcije, ki vedno zahtevajo

kot prvi parameter objekt Command, kamor bi lahko dodali podatke o trenutno prijavljenemu uporabniku. Na tak način bi imeli v samih poslovnih pravilih možnost prebrati podatke o trenutno prijavljenemu uporabniku in preveriti ali uporabnik lahko izvaja akcijo, ki jo je sprožil, ali lahko vidi, kreira, briše oziroma spreminja določeno entiteto itd.

Poleg ogrodja smo razvili tudi reprosto testno aplikacijo, ki demonstrira uporabo orodja. Cilj testne aplikacije je bil testiranje štirih različnih podatkovnih baz. Aplikacija vsebuje štiri teste, ki implementirajo osnovne CRUD operacije. Vsak test vsebuje natanko en testni scenarij, ki pa ga poženemo na vsaki izmed baz posebej. Kot smo že omenili, ogrodje ponuja enoten API za dostop do podatkov, ne glede na tip podatkovnega vira, zato smo v vsakem testu le posredovali vrsto entitete, ogrodje pa je potem samo poskrbelo, da so se v ozadju podatki posredovali v pravilno podatkovno bazo. Na ta način smo prikazali učinkovitost našega aplikacijskega ogrodja, saj je dovolj le nekaj deset vrstic kode in aplikacija poganja štiri teste na štirih različnih podatkovnih bazah, poleg tega pa so baze tudi različnih tipov, saj smo za test uporabili dve relacijski ter dve dokumentni oziroma NoSQL podatkovni bazi.

Razvito ogrodje se lahko enostavno uporabi v produkcijskem okolju, saj so funkcionalnosti, ki smo jih razvili v okviru diplomske naloge, uspešno razvite in preverjene s pomočjo testne aplikacije. Seveda je potrebno vedeti, da ogrodje ni namenjeno razvijanju aplikacij, ki so zelo zahtevne z vidika zmogljivosti in prepustnosti. Od aplikacije, oziroma v našem primeru ogrodja, ki je razvita z namenom posplošitve uporabe različnih delov preko enotnega APIja, ne moremo pričakovati, da bo uporabljal vire karseda učinkovito. Kadar je učinkovitost na prvem mestu, se za to razvije namenska aplikacija, ki iz ciljne podatkovne baze izkoristi najboljše in za ciljno programsko domeno najbolj primerne funkcionalnosti.

Literatura

[1] J. W. Cooper, Java™ Design Patterns: A Tutorial, Boston: Addison Wesley, 2000, str. 141-146.

[2] D. Koenig et al., Groovy in action Second edition, Manning, 2011.

[3] E. Sink, Version Control by Example, Illinois: Pyrenean Gold Press, 2011.

[4] Srirangan, Apache Maven 3 Cookbook, Packt Publishing, 2011.

[5] V. Subramaniam. (2008). Programming Groovy: Dynamic Productivity for the Java Developer, Raleigh: The Pragmatic Programmers, 2008.

[6] P. Tahchiev et al., JUnit in action Second edition, Manning, 2011.

Internetni viri:

[7] ACID: Concurrency Control with Transactions [Online]. Dosegljivo: https://mariadb.com/kb/en/mariadb/acid-concurrency-control-with-transactions/. [Dostopano: 26. 03. 2016]

[8] Maven - Introduction [Online]. Dosegljivo: https://maven.apache.org/what-is-maven.html. [Dostopano: 23. 01. 2016].

[9] Eclipse desktop & web IDEs [Online]. Dosegljivo: https://eclipse.org/. [Dostopano: 23. 01. 2016].

[10] Introduction to Java programming, Part 1: Java language basics [Online]. Dosegljivo: http://www.ibm.com/developerworks/java/tutorials/j-introtojava1/. [Dostopano: 20. 03. 2016].

[11] Lang - Home [Online]. Dosegljivo: https://commons.apache.org/proper/commons-lang/. [Dostopano: 31. 01. 2016].

[12] Learn about Java Technology [Online]. Dosegljivo: https://www.java.com/en/about/. [Dostopano: 18. 02. 2016]

[13] PostgreSQL: About [Online]. Dosegljivo: http://www.postgresql.org/about/. [Dostopano: 23. 03. 2016].